CVS $Revision: 1.7 $Date: 2003/10/14 21:14:19

Physics Simulation Design

G. Bernstein, with Kim/Kushner/Kuznetsova/Gladney/Tucker/Lamoreaux

1. Purpose of the Document

This document will eventually specify all the classes (i.e. Nouns and Verbs) that are needed to implement the desired SNAP simulations. Initially the development will focus on the SNIa investigations, but the class structure will be designed to accomodate future weak lensing and other use cases.

The prerequisites are: the *Simulation Architecture* document (GK), which specifies the system protocols for SNAPSim; and the data flow diagrams (AK) that outline the primary uses for SNAPSim. The old "block diagram"(GB) might closely resemble what is produced here.

In these drafts, **???** mark places that are in need of further work or the opinions of the design group. Objectst that will correspond to a Java class are written in `teletype font`.

2. Domains

It will be useful to divide the physics simulation into different *domains.* These domains do not (yet) correspond to any formal programming structure; they are meant as a way of organizing the problem into nearly separable concepts/tasks, hence will also help in dividing the labor of design/implementation. The guidelines for the definition of domains are:

1. Nouns, or data structures in general, will belong to only one domain. The exceptions are mathematical utility classes.
2. Verbs, or algorithms in general, will sometimes cross domains, *e.g.* by producing Nouns in a new domain from data in previous domains, but the number of involved domains will be kept to a minimum.
3. The branch points for simulations or use cases will tend to be after the completion of a domain. For example, when testing the effect of an instrument specification change, the Universe and Mission domains will be held fixed while we loop over changes to Observatory domain. Tests of different analysis algorithms will start with common SourceData and loop over various Analysis cases.

Following sections define the domains. The data members within each domain are listed in later sections.

Utility: Mathematical/physical classes that are used throughout many domains, such as `WavelengthFunction`, `SphericalCoordinate` or `PSF`.

Universe: everything that happens before the photons hit the telescope or the top of the atmosphere, including `AstronomicalSources`, their distributions in the sky, and intervening effects such as dust and lensing.

Observatory: description of the hardware that converts photons to image data, from the top of the atmosphere to the detector output.

Mission: List of the pointings, times, and configuration of the observatory during exposures, and the algorithm for determining this schedule.

PixelData: the simulated (or real) bits that flow from the spacecraft, along with pixel-level calibrations such as bias and flat-field.

SourceData: the quantities of interest for sources that are measured directly from the images. For points sources, there is `PointSourceDatum` that specifies the position, flux, and uncertainties in these quantities in a single exposure. Spectra and light curves are arrays of `PointSourceDatum` objects. For extended sources, there are additional observables related to shape. There should be no cross-correlation of random errors between different SourceData members, as these are the most basic observable quantities of the objects.[1] SourceData will be in nominally calibrated physical units, discussed further in §8.

Calibration: classes that embody the model used to convert the nominal fluxes in the SourceData into absolute normalizations of the spectra of sources with some assumed spectrum. A simple case is to assign a single photometric zeropoint to each `Channel` (*cf.* §6), but more complex models involve color terms, etc. In other words the Calibration domain takes SourceData for a chosen source and produces an estimate of its absolute flux normalization. There will also astrometric Calibration classes.

Analysis: results of subsequent processing of calibrated SourceData, for instance light curve or spectral parametric fits, fitted supernova models, cosmology fits, lensing maps, etc. Will usually consist of probability distributions over some space of derived quantities, *i.e.* fitted model parameters.

These domains are ordered, in that creation of a Noun in one domain should require only information from previous domains. The primary exception that I can think of is that the Mission scheduling algorithm will end up being forward-looking once we are using partially-analyzed SN data as a spectroscopy trigger (or any other adaptive scheduling technique). Another small "backflow" of information is that when we realize a Universe, we don't want to do the entire sky, just the parts

[1]In fact there might be cross-correlations between properties of objects that overlap on the sky.

that are likely to be observed by a given Mission. So some Mission information must be provided to build the Universe.

3. General Considerations

There are a few design considerations that are applicable to many or all of the domains.

3.1. Truth, Realization, and Model

A full simulation needs to contain multiple code objects to represent the same physical object. There is typically some *truth* version of the object, representing the behavior that is assumed in constructing the artificial measurements. Then there is a *model* version of the object, which is the best estimate of the object's behavior resulting from analysis of the measurements.

Likewise for measurements, there is typically a true value of the expected number of counts (or flux, or position, etc.) and an expected variance—or, more generally, a probability distribution of the measurement. Then there is a *realization* of the measurement, which is a single value drawn from the probability distribution. Equivalently this is the best estimate of the true value of the flux, given the measurement. There is usually an estimate of the variance, which differs from the true variance.

A general goal of our simulation should be that **truth and model of an object, or truth and realization of a measurement, should be embodied by code that is as similar as possible, preferably by the same code.** The interface for `SupernovaIa`, or in fact any `AstronomicalSource` or effect, should be capable of serving the needs of data *simulation* (truth) as well as data *analysis* (model). In a given simulation run, we might analyse the data with a different implementation (*i.e.* model) of `SupernovaIa` for analysis than we used for simulation. But each implementation should be capable of doing either job.

This is important for several reasons:

- In general, a given bit of information, such as a source model, should be coded only once, in order to avoid having two versions that can get out of sync during code development.
- The analysis code development always includes a test that it can properly recover the truth parameters. This is facilitated by having the exact truth model be used by the analysis.
- We want to smooth the transition from simulation pipeline to reduction pipeline; in the latter, there is no known truth.
- Both the simulation and the analysis are essentially the process of modelling the observations—one produces observations from source info, the other vice-versa—so they logically should

share the same code.

Perhaps the only significant consequence of this guideline is that models should be capable of providing the derivatives of observables with respect to model parameters. Such derivatives greatly speed the analysis phase.

Another guideline in this area is that **truth and model (or truth and realization) should be stored in distinct instances of the model class.** Any correspondence between truth and model instances should be recorded in an external catalog class, not within either the truth or the model class. There will be many simulation runs in which there is, for example, truth data but no realization (*e.g.* in a Fisher analysis), and many pixel-based simulation runs in which there are model objects that have no corresponding truth (*e.g.* false positives or a blind analysis). Of course the latter is true for the real data as well!

3.2. Fundamental Operations

A SNAPSim task typically involves four general tasks:

1. Specify the Universe being observed, the Observatory being used, and the Mission strategy, then realize each of these.
2. Determine which targets of interest were observed at what times with what hardware channels.
3. Realize the SourceData for the each observation of each target (with or without realizing PixelData).
4. Analyze (and perhaps calibrate) the SourceData to obtain final quantities of interest, typically the uncertainty on dark energy parameters.

A goal of this design is to make each of the first three processes as generic as possible, in the sense that our data structures can flexibly handle different kinds of observatories, astronomical objects, spectral vs imaging data, etc., without having to recode the Verbs that do these steps. This means using polymorphism and interfaces wisely.

The following subsections list some operations that seems very basic to the simulation process, so many of the interfaces will be designed to work smoothly with these Verbs/facilities. **??? Are these going to be verbs?**

3.2.1. Astrometric Calculator

In Step 2, it is necessary to determine which sources are projected within the borders of a given detector in the focal plane. This requires some efficient means of calculating the sky-coordinate boundaries of the detector given the optical distortions, the pointing and orientation of the telescope, and distortions induced by atmospheric and gravitational lensing. We need to develop an Astrometric Calculator which can implement these calculations efficiently for a wide variety of distortion effects that are distributed between the Universe, Observatory, and Mission domains.

3.2.2. Exposure Time Calculator

Step 3, realization of SourceData, is a fundamental operation, and is essentially the job of the Exposure Time Calculator (ETC) when we are not simulating PixelData. For Fisher analyses, we will ask the ETC only to calculate the variance of the measurement, and we do not need the realization itself. The ETC is another facility that needs to be efficient and flexible.

The ETC, to realize a point-source measurement, needs to know from different domains:

- From the Mission:
  - exposure time
  - exposure repetition pattern (dithering, etc)
- From the Observatory:
  - pixel pitch and plate scale
  - detector noise model
  - cosmic ray model
- From combination of Universe and Observatory:
  - target count rate (integrated over band)
  - background count rate (integrated over band)
  - ePSF (effective PSF, includes all optical & detector contributions)

For extended-source analysis such as galaxy photometry or shape analysis, also needs to know a galaxy `Shape`, from the Universe domain.

So these are all quantities that we should be prepared to derive for any type of astronomical target and any type of observatory setup.

3.2.3. Image Simulator

The exact same information that the ETC needs is also required to create pixel-level simulations. A fast and flexible "draw" routine will be needed, which can execute distortions and convolutions of the intrinisic source shapes.

3.2.4. Model Fitting

Most Analysis activities, and many Calibration activities, will involve fitting a parametric model of the sources, dust, cosmology, and instrument calibration to a series of SourceData or higher-level data. **???** We need to work out a syntax for flexibly specifying parameter sets that consist of elements of various Nouns, for specifying figure-of-merit functions for optimizing, and for holding covariance matrices or likelihood functions of resultant fits.

4. Utility Classes

4.1. Spectral Information

We will be awash in functions of wavelength: source spectra, extinctions, background brightnesses, QE's, etc., that have to be multiplied and integrated. A uniform interface is needed:

```
interface WavelengthFunction {
  double          valueAt(double lambda);
  void            validity(double& lambdaMin, double& lambdaMax);
  void            support(double& lambdaMin, double& lambdaMax);
  SpectralUnits   units();
  double          resolution();
  ???             preferredValues();
}
```

Each implementation embodies some function $f(\lambda)$, which is defined in the region of `validity` $[\lambda_{\min}, \lambda_{\max}]$. Requesting the `valueAt` a $\lambda \notin [\lambda_{\min}, \lambda_{\max}]$ will throw an exception. **??? define exception class.**

$f(\lambda)$ is gauranteed to be zero outside the interval of `support`. Filter functions will have the support region contained within the definition region, but source spectra may be non-zero outside the bounds of their definition.

The `resolution` and `preferredValues` are hints to the `SpectrumIntegrator`: the former suggests the largest allowable $\ln\lambda$ step that will capture the variation of the function, and the latter may contain information about the nature of a tabulated function.

4.1.1. Units

The functions carry a tag to indicate their units. This is a Noun (**??? internal representation TBD**) that we design and serves as a type-safety mechanism, for instance insuring that a QE is not used where a source spectrum is required.

```
class SpectralUnits extends Noun {
private:
???
public:
     bool equals(SpectralUnits rhs) const;
     SpectralUnits times(SpectralUnits rhs) const;
     static final SpectralUnits Luminosity;
     static final SpectralUnits Flux;
     static final SpectralUnits Area;
     static final SpectralUnits SolidAngle;
     static final SpectralUnits SurfaceBrightness;
     static final SpectralUnits TransferFunction;
     static final SpectralUnits Rate;
}
```

The units of these quantities should be standardized throughout the simulation. The most robust units for luminosities are photons per ln(wavelength) per second—luminosities expressed this way are independent of energy or wavelength units; the value is the same as per ln(frequency); evolve only as $(1+z)$ under cosmological redshift because the photon count and logarithmic intervals are invariants; and convert naturally to e in the detectors. If we adopt μm as the wavelength unit, m^2 as the collecting-area unit, and sr as the solid-angle unit, then the SpectralUnits constants are then as given in Table 1. The `TransferFunction` units are appropriate for extinction, atmospheric/atmospheric/filter transmission curves, and QE curves.

4.1.2. Implementations of `SpectralFunction`

We will have numerous implementation of the `SpectralFunction` interface. These will certainly include

```
class ScalarSpectralFunction extends Noun implements SpectralFunction;
```

```
class TabularSpectralFunction extends Noun implements SpectralFunction;
```

```
class BoxcarSpectralFunction extends Noun implements SpectralFunction;
```

```
class ProductSpectralFunction extends Noun implements SpectralFunction {
private:
    NounArray factors;
public:
    ????
}
```

The `ProductSpectralFunction` is defined to be the product of all its factors, each of which is a `SpectralFunction`. For the `ProductSpectralFunction`, the `support` and `validity` ranges are the intersections of all its factors'. The resolution is the minimum of its factors'. The `factors` array will hold **??? links to** the component functions. There will be methods for adding factors, etc.

4.1.3. The Integrator

A final class in this set is the utility to integrate a `SpectralFunction`. This can be a Verb **???**.

```
class SpectrumIntegrator implements Verb {
    <required Verb stuff...>
    double integrate(SpectralFunction f, SpectralUnits units);
}
```

The `SpectrumIntegrator` returns

$$\int_{\lambda_{\mathrm{min}}}^{\lambda_{\mathrm{max}}} f(\lambda)\, d\ln\lambda \tag{1}$$

with the integral over the `support` of the `SpectralFunction` f. An exception is thrown if the `validity` does not span the `support`. The units of the integral quantity are output. They will tend to be `SpectralUnits::Rate`.

The `SpectrumIntegrator` should be intelligent enough to use `f.resolution()` and to extract the scalar quantities from a `ProductSpectralFunction` before integration.

It will be desirable to have `SpectrumIntegrator` cache the results of its integrations, because the same ones will likely be repeated many times, *e.g.* the flux of a supernova at peak integrated over the B band.

4.2. Astrometric Coordinates and Maps

Positions on the sky will need to be specified by an abstract class

```
class SphericalCoordinates extends Noun {
private:
    double x,y,z;   //Three direction cosines
    abstract void convertToICRS(double x, double y, double z);
    abstract void convertFromICRS(double x, double y, double z);
public:
    ????
}
```

Derived classes are then constructed for each specific coordinate system:

```
class SphericalCoordinatesICRS extends SphericalCoordinates;
class SphericalCoordinatesEcliptic extends SphericalCoordinates;
class SphericalCoordinatesGalactic extends SphericalCoordinates;
class TangentPlaneCoordinates extends SphericalCoordinates;
```

The last of these specifies spherical coordinates as a tangent-plane projection about a chosen axis on the sky. It requires a class

```
class Orientation extends Noun {
    SphericalCoordinates axis;
    double  positionAngle;
}
```

Coordinate mappings will arise in both the Universe and the Observatory domains, and will be modelled or used in the Calibration and Analysis domains. There will be a coordinate-transform class:

```
interface Distortion {
  void forward(SphericalCoordinates xyTrue, SphericalCoordinates& xyObs)
  void inverse(SphericalCoordinates xyObs,  SphericalCoordinates& xyTrue)
  Matrix magnification(SphericalCoordinates xyObs);
}
```

The `magnification()` method gives the local magnification matrix, *i.e.* the linear expansion of the distortion map about a given line of sight.

The implementations of `Distortion` will include some gravitational lensing cases [**???** lensing distortion depends upon the redshifts of the source and lens; we need some way to incorporate this] from the Universe; a map of the optical distortions of the telescope about the optic axis; and perhaps additional classes

```
class StellarAberration extends Noun implements Distortion {
private:
   SphericalCoordinates velocityApex;
   double               beta;
...
}
class AtmosphericRefraction extends Noun implements Distortion {
}
```

There is an implementation that is the composition of several `Distortions`:

```
class DistortionProduct extends Noun implements Distortion;
```

unlike `PSF` and `SpectralFunction` compositions, the order of `Distortions` is significant.

??? The domain of `Distortion` functions needs a little work. For a given Observatory `Channel` we will want a `Distortion` that is actually a map from pixel coordinates on the focal plane (in μm probably) into a `TangentPlaneCoordinate` which is in radians but is relative to the optic axis. The `Orientation` of the optic axis then turns this into a map into `SphericalCoordinatesICRS`.

A region on the sky can be described through this interface:

```
interface SolidAngle {
  bool includes(SphericalCoordinates point);  //true if region includes the point
  bool overlaps(SolidAngle rhs);  //true if region touches another region
  bool includes(SolidAngle rhs);  //true if region includes another region
  double aread();               // returns steradians enclosed
}
```

Instead of being an interface with implementations of many kinds of regions, we might just restrict ourselves to polygonal regions. Otherwise the coding of `overlaps` and `includes` could be messy.

4.3. Time

A class for a UT date and time:

```
class UT extends Noun {
  double interval(UT rhs);   //time interval (s) between this and rhs
}
```

4.4. Point Spread Functions

The blurring of the astronomical image before and during detection is described by a point spread function (PSF):

```
interface PSF {
  complex kValue(double kx, double ky);
  complex xValue(double x, double y);
  double  maxK();          //largest significant k-vector
  double  stepK();                       //suggested k-space resolution
  bool    isAxisymmetric();
  // Other calls that specify the spatial extent, k-space extent,
  // and various moments.
}
```

We can expect to require several implementations

```
class PSFGaussian extends PSF;
class PSFAiry extends PSF;
class PSFKolmogorov extends PSF;
class PSFBox extends PSF;
class PSFTabularK extends PSF;
class PSFProduct extends PSF;
}
```

The Gaussian is a useful analytic PSF, and accurately describes the charge diffusion contribution. The Kolmogorov form arises from atmospheric seeing; the Box is the typical effect of pixelization; `PSFTabularK` allows a PSF that is tabulated in k space.

The last implementation of this interface is a PSF that consists of a product of other PSFs in a NounArray. It can propagate `maxK()`, `stepK()` and `isAxisymmetric()` in the expected fashion. A clever version would be capable of recognizing efficent analytic combinations, for example all `PSFGaussian` contributions can be combined into a single one.

??? The units of PSFs need to be specified. Normally the domain of a PSF measured in arcseconds or radians. For detector effects, however, the domain is sometimes specified in microns. Perhaps we can standardize on radians, and leave it to the `Detector` classes to take a plate scale and provide a `PSF` that is in radian units. The output units of a PSF are usually such that the integral over its domain is unity (value at $k = 0$ is unity). A pixel response function (PRF), however, is a PSF that specifies the conversion from incident intensity ($\gamma\,\mathrm{s}^{-1}\,\mathrm{sr}^{-1}$) to the count rate in a pixel ($\gamma\,\mathrm{s}^{-1}$), so that its integral must have units of inverse sr.

4.5. Parameter Sets

Fitting of a model to a set of observations is going to be an oft-used process. A standardized representation of the parameters, data, likelihood functions, and uncertainties should be implemented.

??? Some very preliminary ideas:

```
class ParameterVector extends Noun {
public:
  void setElement(int i, double v); //access parameter i
  double getElement(int i);
  bool isLinear(int i);         //are observables linear in this parameter?
  int size();
}

// Classes describing measured parameters and likelihood thereof
class ObservableVector extends Noun;   // measurements to be fit
interface Likelihood {
  double logLikelihood(ObservableVector o);
  Vector logLDerivatives(ObservableVector o);
}
class GaussianLikelihood extends Noun implements Likelihood {
private:
  PositiveDefiniteMatrix covariance;
}

// An interface that defines a model being fit:
interface Model {
ObservableVector predict(ParameterVector p);
Matrix parameterDerivatives(ParameterVector p);
}

class ModelFit implements Verb {
void fit(Model m,
         ObservableVector o,
 Likelihood l,
         ParameterVector p,
         Covariance pc);
}
```

```
class FisherAnalysis implements Verb {
PositiveDefiniteMatrix fisherMatrix(Model m,
                                    ParameterVector p,
                                    Likelihood l)
}
```

The `ModelFit` is our fitting routine, maybe there will be several such as Marquardt-Levenberg, downhill simplex, linear, etc. **???**The `Covariance pc` that is output to describe the error in the fit presumes a Gaussian distribution; in some cases we might need to return a more general `Likelihood` over the parameters.

To fit a model, one must create an `ObservableVector` and a `LikelihoodModel` that relects the uncertainties in the observables. Then one must create an implementation of `Model` that knows how to distribute the variables in the `ParameterVector` to all the physics models, run the necessary Verbs to predict the observables, and calculate derivatives w.r.t. parameters.

The initial `ParameterVector` guess is fed to the `ModelFit` Verb along with the information on model and observables. **???** How does the `Model` inform the `ModelFit` which parameters enter linearly into the observables? This is important for efficient fitting.

5. Universe Domain

5.1. Cosmology

Describes the zeroth-order geometry of the Universe.

```
interface Cosmology {
  // generic methods for changing cosmological parameters
  void   setParameters(ParameterVector p);
  ParameterVector getParameters();

  double dA(double z);    //angular diameter distance
  double dL(double z);    //luminosity distance
  double dVdz(double z);  //volume element
  double lookback(double z); //lookback time

  Vector dDAdp(double z);  //derivatives w.r.t. parameters
  Vector dDLdp(double z);  //derivatives w.r.t. parameters

  // others for linear growth factor, etc.???
}
```

One implementation is of course our usual cosmology with parameters $\{\Omega_m, \Omega_X, w, w_a, h\}$.

???Should the `Cosmology` also be responsible for inhomogeneities (lensing), so that the above calls take a `SphericalCoordinate` as argument as well?

5.2. Astronomical Sources

Any source of photons in the Universe.

```
interface AstronomicalSource {
  // generic methods for changing model parameters
  void   setParameters(ParameterVector p);
  ParameterVector getParameters();

  SpectralFunction luminosity(UT t); // rest-frame luminosity
  SpectralFunction dLdp(UT t);       // luminosity deriv w.r.t. parameters
  ??? dLdt ???

  SphericalCoordinates  position(UT t);  //position before any distortion
  double redshift();   // cosmological redshift (if cosmological, 0 otherwise)
  double distance(UT t);  // distance as function of time (if local)
  double velocity(UT t);  // rest-frame velocity

  // Some helpful hints:
  bool isVariable();     //Does luminosity vary w/time
  bool isMoving();       //Does (unlensed) position change with time

  Shape  itsShape(double lambda);   //Shape (in proper units)
}
```

??? A note on Dates: I suggest a convention that the date input to the `AstronomicalSource` (or indeed to any Universe class) be the date at which the light arrives at the Solar System barycenter. The `AstronomicalSource` is then responsible for calculating the proper time at the source, which it can do because it knows the redshift z. A complication is that gravitational lensing introduces time delays; but only in very rare cases will we care about this, such as when simulating multiply-imaged sources, and we can plan perhaps to worry about these cases individually.

5.2.1. Shapes

Note the introduction of another class that describes a map of surface brightness over 2d angular or physical variables:

```
interface Shape {
  complex kValue(double kx, double ky);
  complex xValue(double x, double y);

  // Some hints for usage of this Shape:
  double  maxK();          //largest significant k-vector
  double  stepK();                       //suggested k-space resolution
  bool    isAxisymmetric();

  // additional methods for "draw" onto an image, returning
  // particular moments, etc.
}
```

Expected implementations of `Shape` include

```
class PointSource extends Noun implements Shape;
class GaussianEllipse extends Noun implements Shape;
class ExponentialEllipse extends Noun implements Shape;
class DeVaucouleursEllipse extends Noun implements Shape;
class Shapelet extends Noun implements Shape;
class TabulatedShape extends Noun implements Shape;
```

There will probably have to be a Verb that can convolve a `Shape` with a `PSF` to return another `Shape`:

```
class ConvolveShape implements Verb {
  Shape convolve(Shape source, PSF thePSF);
}
```

5.2.2. Supernova Classes

A base class for all supernova would be:

```
class Supernova extends Noun implements AstronomicalSource {
public:
```

```
  UT dateOfExplosion();
  bool isVariable() {return true;}
  bool isMoving() {return false;}
  Galaxy itsHost() {return hostGalaxy.target()};
  Shape itsShape(double lambda) {return PointSource;}
private
  Link hostGalaxy;
}
```

??? Note use of the SNAPSim `Link` class here to hold a pointer to the host `Galaxy` object. The syntax used here for extracting the target of the link is a guess at what SNAPSim will provide. For most of this document I have glossed over the distinction between `Link` and `Composition` relationships.

Type Ia supernovae do not in general have any methods or data beyond those of this `Supernova` class. But we may choose to implement, as in SNAPfast, models of TypeIa that predict some intermediate observable quantities other than just the `luminosity`, *e.g.* light-curve parameters such as stretch, rise time, etc. So there might be an additional class (or an abstract class with several implementations) called

```
class SupernovaIa extends Supernova {
  ... // Implements all the methods of AstronomicalSource() and
  ... // Supernova()

  ObservableVector observables();  //intermediate observables
  ObservableVector dObsdp(int i);  //and their derivs w.r.t. parameter i
}
```

5.2.3. Galaxies

Galaxies are `AstronomicalSources` with finite extent. The spectrum of a galaxy would likely be specified by some set of star-formation and extinction parameters. Note that `luminosity` should return the total luminosity of the entire galaxy, `itsShape` returns the distribution of that luminosity across the sky at a chosen wavelength.

??? Note that a `Galaxy` will need to know the `Cosmology` somehow in order to calculate its `luminosity`, because the stellar-evolution codes will need to know the age of the Universe at the relevant redshift.

```
class Galaxy extends Noun implements AstronomicalSource {
public:
```

```
  bool isVariable() {return false;}
  bool isMoving() {return false;}
  ??? some kind of connection with a catalog entry
}
```

5.2.4. Stars

Assuming here that all stars are too close to have comsological redshifts, there is a base class that looks like

```
class Star extends Noun implements AstronomicalSource {
public:
  Shape itsShape(double lambda) {return PointSource;}
  double redshift() {return 0.;}
private:
  double distance;
  ??? proper motion specification
}
```

Some stars are variable, some aren't. We'll assume that `isMoving()` is true only if there is non-zero proper motion; parallax does not count as "moving."

5.3. Propagation Effects

Any medium that lies between the `AstronomicalSource` and the `Detector` may be derived from

```
class PropagationEffect extends Noun {
public:
  void   setParameters(ParameterVector p);
  ParameterVector getParameters();

  SpectralFunction transmission();        //Transmission of bgrnd photons
  SpectralFunction emission();          //surface brightness of added photons
  Distortion distortion();   //distortion of bgrnd image
  PSF blur();                           //blurring of bgrnd image

  // This one needed for modelling:
  SpectralFunction dTdp(int i);           //Deriv of transmission  w.r.t. parameter
```

```
}
```

In general an intervening `PropagationEffect` can have four effects on our view of its background: absorbing photons; adding photons; distorting the image; or blurring the image. Galactice dust, for example, has important extinction but no significant visible emission, distortion, or blurring, so it might be realized as a `PropagationEffect` that has null returns for `emission(), distortion(), and blur().` The atmosphere is probably significant in all four aspects (this will be in the Observatory domain, not Universe).

Each individual observation of an `AstronomicalSource` will be done through a series of `PropagationEffects`, the effects of which can be compounded because of our ability to generate products of `SpectralFunctions`, `Distortions`, and `PSFs`.

The important `PropagationEffects` in the Universe domain will be:

```
class HostGalaxy extends Noun implements PropagationEffect;
class IntergalacticDust extends Noun implements PropagationEffect;
class GalacticDust extends Noun implements PropagationEffect;
class GravitationalLensing extends Noun implements PropagationEffect;
class ZodaicalLight extends Noun implements PropagationEffect;
class TopOfAtmosphere extends Noun implements PropagationEffect;
```

All of these may be considered to have null `blur()` in the visible/NIR. Non-null characteristics include:

`HostGalaxy` has `transmission()` as a `Dust` screen at the redshift of the host. The `emission` is the surface brightness of the host galaxy.

`IntergalacticDust` has `transmission()` specified by some model of grey dust and the redshift of the source.

`GalacticDust` has `transmission()` as a `Dust` screen at zero redshift.

`GravitationalLensing` has only `distortion()`, which is determined by the source redshift and position.

`ZodaicalLight` has only `emission()`, which is determined by the ecliptic coordinates of the observation.

`TopOfAtmosphere`This is just a placeholder entry. All of the methods return null, but it allows us to mark the point in the stack of `PropagationEffects` which divides the Universe from the Observatory. It is the job of the Calibration domain to simulate the spectral response of elements past this marker.

Classes that will certainly be needed to implement the astronomical `PropagationEffects` are

```
// Clayton-Cardelli-Mathis model of extinction:
class CCMDust extends Noun implements SpectralFunction {
private:
  double Av, Rv;     // A_V and R_V of dust
  double z;          // Redshift of the dust screen
}

// Greg Aldering's model for Zodiacal emission spectrum
class ZodiacalLight extends Noun implements SpectralFunction {
private:
  double solarElongation;
  double eclipticLatitude;
}

// A model for grey dust absorption
class GreyDust extends Noun implements SpectralFunction {
private:
  double sourceZ; // redshift of source
  double ???            // parameters of the dust distribution model
}
```

??? Maybe the Zodi goes with observatory, since it depends upon the position of the observatory at time of observation.

5.4. Distribution Functions

Distribution functions tell us how to populate the Universe with `AstronomicalSources` and what `PropagationEffects` their photons will encounter before entering the Observatory. We first have an interface for generating a list of sources that might be seen over a given time period in a given part of the sky. **???** I think the distribution functions want to be Verbs, so that the `run()` method generates the desired source list. I'm not sure in Java if you can have a heirarchy of interfaces or if I need to make this an abstract base class.

```
class SourceGenerator extends Noun implements Verb {
public:
  NounArray generate(SolidAngle a,
                     UT startTime, UT endTime,
     UniformDeviate u);
```

```
}
```

The `UniformDeviate` object is a random number generator that is passed in. We assume a model in which the entire simulation run shares a single instance of the random number generator.

There can be various implementations of `SourceGenerator`, which make SNe, galaxies, stars, or perhaps jointly generate the galaxies and SNe so that the host relationships are physically based. Some implementations would have additional parameters, such as the min/max redshift of interest, or the faintest magnitude of galaxy to generate. A given realization of the Universe can have one or more implementations of `SourceGenerator` instantiated. There should be a master list of all active `SourceGenerators`:

```
class MasterGenerator extends Noun implements Verb {
public:
  NounArray sourceGenList;
  NounArray propagationGenList;
  NounArray generate(SolidAngle a,
                     UT startTime, UT endTime,
     UniformDeviate u);
}
```

The `generate()` method of this class just calls `generate()` for all of the `SourceGenerators` in the `sourceGenList`.

It is also necessary to realize the `PropagationEffects`, for example create a lensing or extinction map on the sky. This is the job of a `PropagationGenerator`. For example there is

```
class SchlegelDustMap extends Noun implements PropagationGenerator {
public:
  PropagationEffect generate(SphericalCoordinate s);
}
```

which can produce a `CCMDust` extinction law for the chosen coordinates based on the Schlegel/Finkbeiner/Davis map.

???I'm not sure how to handle this: the propagation effects can depend upon observing circumstances, for example the zodi depends upon the time of year of observation. Also the sources might be "born" with certain propagation effects attached, for example the host dust, which is particular to a SNe and is not determined from a global map or function. Need to figure out how to do this.

6. Observatory Domain

7. Mission Domain

8. SourceData Domain

9. Calibration Domain

The task of the calibration is to create a model for the response of the instrument. Typically the "instrument" is taken to be everything from the top of the atmosphere down (or from the entrance aperture of a space telescope). In essence the Calibration domain thus contains, for each `Channel` of the detector, a little model of how the positional and flux outputs of that `Channel` are related to the position and flux of the celestial object—in other words, a `Distortion` plus a `SpectralFunction` that describes the absolutely-calibrated effective collecting area of the `Channel`.

[**NOTE:** The following expository material is for discussion purposes, probably won't go into the actual specification document.]

The simulation code in the Observatory domain contains information that fully describes the instrument response. But this *truth* information should not be available to the Analysis domain because when analyzing the real data, we will not have perfect *a priori* knowledge of the instrument response. We will have to create a model, using the data itself, information from ground tests, and calibration observations with the telescope. So the Calibration model of the instrument, unlike the Observatory specifications, have free parameters that have to be adjusted to give the best estimate (and uncertainties) that agrees with all calibration information.

Flux calibrations can be used two different ways. One is to have a model for the instrument's spectral response. A posited input flux spectrum (*i.e.* at the top of the atmosphere) can be convolved with the response model to *predict* the flux in the `SourceData`, which are our instrumental outputs. Most rigorously we then view the analysis as a process of adjusting the free parameters in the source models *and* in the calibration models until we obtain best agreement with the collection of `SourceData` from all the science and calibration observations. Let's call this Method A.

Typical astronomical calibrations often try to do something different, let's call it Method B: they posit a set of *standard bandpasses* and try to come up with a conversion from the instrumental outputs into fluxes through the posited standard bandpasses. The typical photometric calibration equation takes intrumental magnitudes in one or more bands (plus perhaps engineering data such as airmass), maybe some prior info on the spectrum/color of the target, and outputs an estimate of fluxes in standard bands/colors.

Method A is preferred for analysis of the experiment's data. Ultimately we want a model of the whole system (Universe plus instrument) that best reproduces the data. Whenever we have a model for the spectra of the sources (*e.g.* SN Ia's, galaxies of different type/redshift) we want to be able to analyze things this way.

Method B is preferred when we need to compare the data to external systems, or when we want to place all the data on the same system even though the instrument properties vary (*e.g.* for different airmasses, two slightly different QE curves or filters in same nominal band, etc.). It also has the advantage that the Analysis domain classes can just fit their source models to the standard-mag outputs, and don't ever have to examine the instrumental fluxes directly.

Figures 1 show the inputs and outputs of the calibration process in both Methods. In Method A, the calibration parameters (which specify the model passbands) are combined with a posited source spectrum to give a model instrumental output for each `Channel`. In Method B, the instrumental fluxes are *input*, along with the calibration parameters, and the output are estimated standard-band fluxes. Input includes uncertainties on the instrumental fluxes, which are propagated to a covariance matrix for the output fluxes.

I am reasonably sure that we will need to implement Method A. But astronomical calibration people are not accustomed to having to estimate passband functions, they usually just deal with moments of the passbands (like color terms, which are essentially 1st moments of the passband). There must be some way to bridge the gap between these two views of calibration, and come up with a uniform interface that can suit both. But I have not thought of it yet.

In any case we will need these kinds of classes:

```
class ParameterVector extends Noun;     //The parameters of the Calibration model
class ParameterCovariance extends Noun; //Uncertainties on the Calibration model
class CalibrationModel;    // ???? The A or B method black box itself
```

There are also classes that the calibrators themselves will use to refine the calibration model, using observations of standards, multiple observations of the same targets, etc.

The `ParameterVector, ParameterCovariance` classes will likely be defined in the Utility domain.

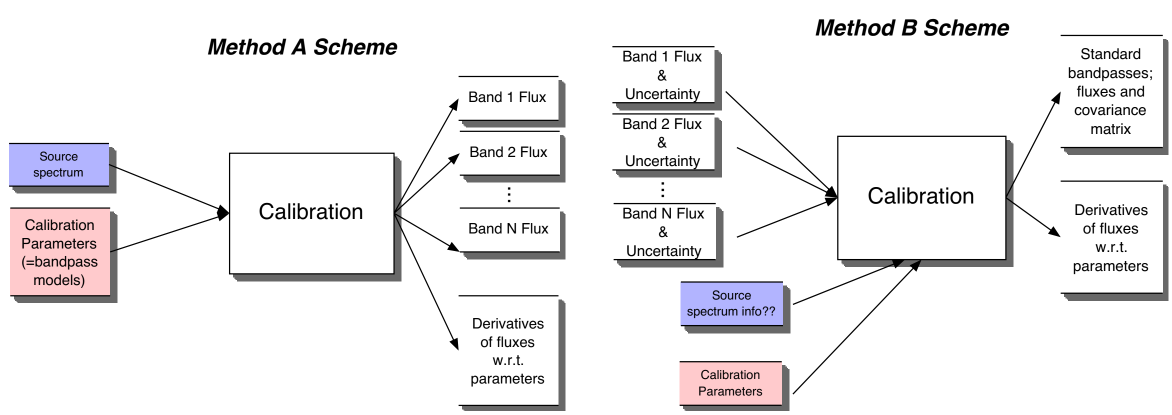

Fig. 1.— Two possible views of a Calibration process.

10. Analysis Domain

Table 1. SpectralUnits Definitions

Quantity	Units
Luminosity	$\gamma\ (\ln\lambda)^{-1}\ \mathrm{s}^{-1}$
Flux	$\gamma\ (\ln\lambda)^{-1}\ \mathrm{m}^{-2}\ \mathrm{s}^{-1}$
Area	m^2
SolidAngle	sr
SurfaceBrightness	$\gamma\ (\ln\lambda)^{-1}\ \mathrm{m}^{-2}\ \mathrm{s}^{-1}\ \mathrm{sr}^{-1}$
TransferFunction	(scalar)
Rate	s^{-1}
λ	μm